Where We Are In The Course

- Basic Java (review)
- Software Design (Phone Directory)
- Correctness and Efficiency:

  Exceptions, Testing, Efficiency (Big-O)
- Inheritance and Class Hierarchies
- Lists and the **`Collection`** Interface

  Building Block for Fundamental Data Structures
- Stacks: Perhaps the Simplest Data Structure
- Queues: The Second Simplest

Queues

Based of Koffmann and Wolfgang
Chapter 6

Chapter Outline

- Representing a waiting line, i.e., *queue*
- The methods of the **`Queue`** interface:

  `offer`, **`remove`**, **`poll`**, **`peek`**, and **`element`**
- Implement the **`Queue`** interface:
  - Singly-linked list
  - Circular array (a.k.a., circular buffer)
  - Doubly-linked list

Chapter Outline (2)

- Applications of queues:
  - Simulating physical systems with waiting lines ...
  - Using Queues and random number generators

The Queue Abstract Data Type

- *Visualization:* queue = line of customers waiting for some service
- In Britain, it is the common word for this
- Next person served is one who has *waited longest:*
  - First-in, First-out = FIFO
- New elements are placed at the *end of the line*
- Next served is one at the *front of the line*

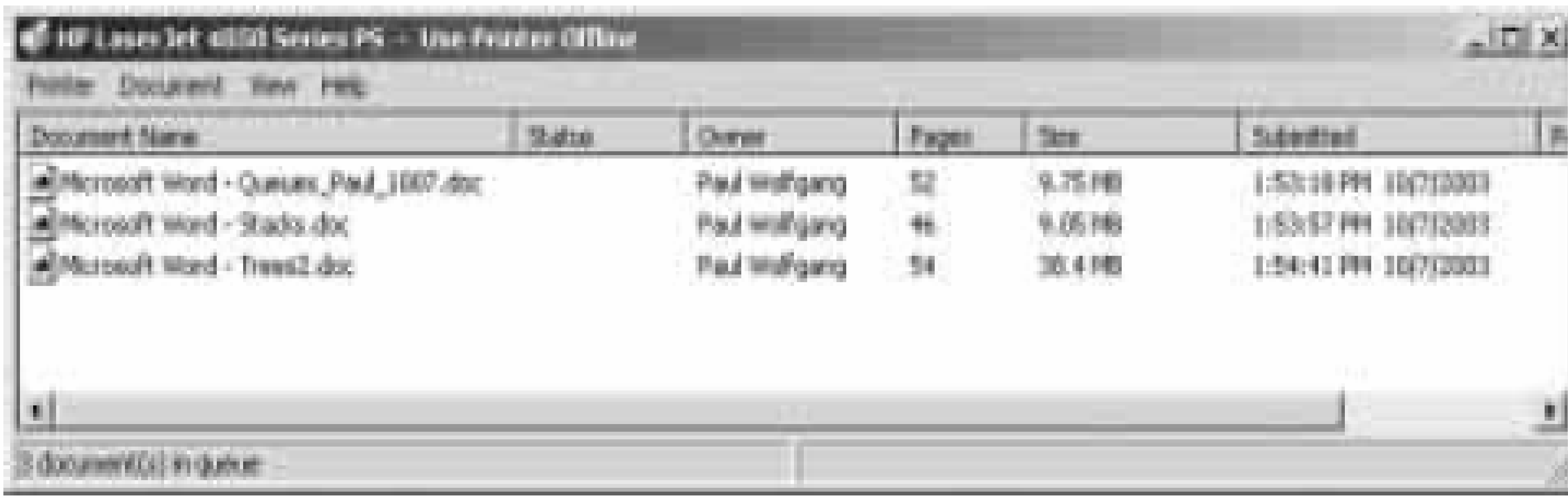
A Print Queue

- Operating systems use queues to
  - Track tasks waiting for a scarce resource
  - Ensure tasks carried out in order generated
- Print queue:
  - Printing slower than selecting pages to print
  - So use a queue, for fairness
- A stack would be inappropriate (more in a moment)
- (Consider multiple queues, priorities, etc., later)

A Print Queue (continued)

FIGURE 6.2

A Print Queue in the Windows Operating System



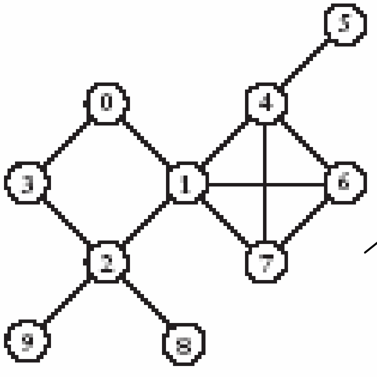
Unsuitability of a Print *Stack*

- Stacks are last-in, first-out (LIFO)
- *Most recently selected* document would print next
- Unless queue is empty, your job *may never execute*
  - ... if others keep issuing print jobs

Using a Queue to Traverse a Multi-Branch Data Structure

- Graph models network of nodes
  - Several links may connect each node to other ones
- A node in the graph may have several successors
- Can use a queue to ensure that nodes closer to the starting point are visited before nodes that are farther away

FIGURE 6.4
A Network of Nodes



Illustrates a ***breadth-first*** traversal of the network of nodes

Specification of the `Queue<E>` Interface

`boolean offer (E e)`

Insert e at rear of queue; return true if worked

`E remove ()`

Remove and return front entry; exception if none

`E poll ()`

Remove and return front entry; null if none

`E peek ()`

Return front entry without removing; null if none

`E element ()`

Return front entry without removing; exception if none

Specification of the `Queue<E>` Interface (2)

- Part of the **`Collection`** hierarchy, so ...
- Offers many other methods, including:
  - **`add`**
  - **`size`**
  - **`isEmpty`**
  - **`iterator`**

The Java API Implements `Queue<E>`

- Easy to implement queues with doubly-linked lists
- Class **`LinkedList<E>`** implements **`Queue<E>`**
- Example use:

```
Queue<String> names =
      new LinkedList<String>();
```

Maintaining a Queue of Customers

Problem: Write a menu-driven program that:

- Maintains a list of customers waiting for service
- Can add a new customer at the end
- Can display name of next customer to serve
- Can display the length of the line
- Can determine how many people are ahead of a particular waiting customer

Maintaining a Queue of Customers (2)

Analysis: Because service is in FIFO order, a queue is the appropriate data structure

- Top level algorithm: while user not finished,
  - Display menu and obtain choice, then
  - Perform the selected operation
- The operations are all basic queue operations ...
- *except:* obtaining a customer's queue position

Maintaining a Queue of Customers (3)

Obtaining a customer's queue position:

- Use an *iterator* to produce waiters in order
- Count until we find the customer
- Handle the *not found* case, too

Customer Queue: Common Variables

```
private Queue<String> customers;
...
customers = new LinkedList<String>();

String name;
```

Customer Queue: Add Customer

```
name = JOptionPane.showInputDialog(
  “Enter new customer name”);

// add to end (rear) of the queue
customers.offer(name);
```

Customer Queue: See Who Is Next

```
// Note: throws exception if queue empty
name = customers.element();

JOptionPane.showMessageDialog(null,
  “Customer “ + name + “is next”);
```

Customer Queue: Remove Next

```
// Note: throws exception if queue empty
name = customers.remove();

JOptionPane.showMessageDialog(null,
  “Customer “ + name + “is now removed”);
```

Customer Queue: Size

```
int size = customers.size();

JOptionPane.showMessageDialog(null,
   “Size of queue is: “ + size);
```

Customer Queue: Customer Position

```
name = JOptionPane.showInputDialog(
    “Enter customer name”);
int cnt = 0;
for (String inQ : customers) {
  if (inQ.equals(name)) {
    JOptionPane.showMessageDialog(null,
      “# ahead of “ + name + “: “ + cnt);
    break;
  } else { ++cnt; }
}
if (cnt == customers.size())
  JOptionPane.showMessageDialog(null,
    name + “ is not in the queue”);
```

Customer Queue: Empty Queue

```
try {

  ... switch on desired operation ...

} catch (NoSuchElementException e) {
  JOptionPane.showMessageDialog(null,
    “The queue is empty”, “”,
    JOptionPane.ERROR_MESSAGE);
}
```

Implementing Queue: Doubly-Linked Lists

This is a simple *adapter class*, with following mappings:

- Queue **`offer`** maps to **`addLast`**
- Queue **`poll`** maps to check **`size`** then **`remove`**
- Queue **`peek`** maps to check **`size`** then **`getFirst`**
- ...

It should be easy to see how DL Lists easily do queues:

- Insert at last
- Remove at first

Implementing Queue: Singly-Linked List

This requires **`front`** and **`rear`** **`Node`** pointers:

```
public class SLLQueue<E>
    extends AbstractQueue<E>
    implements Queue<E> {
  private Node<E> front = null;
  private Node<E> rear  = null;
  private int size = 0; // avoid counting
...
}
```

Implementing Queue: Singly-Linked List (2)

- Insert at tail, using **`rear`** for speed
- Remove using **`front`**
- Adjust **`size`** when adding/removing
  - No need to iterate through to determine **`size`**

Implementing **SLLQueue**

```
public boolean offer (E e) {
  Node<E> node = new Node<E>(e);
  if (front == null) {
    front = node;
  } else {
    rear.next = node;
  }
  rear = node;
  ++size;
  return true;  // added item e
}
```

Implementing **SLLQueue** (2)

```
public E poll () {
  E e = peek();
  if (e != null) {
    front = front.next;
    size--;
  }
  return e;
}
```

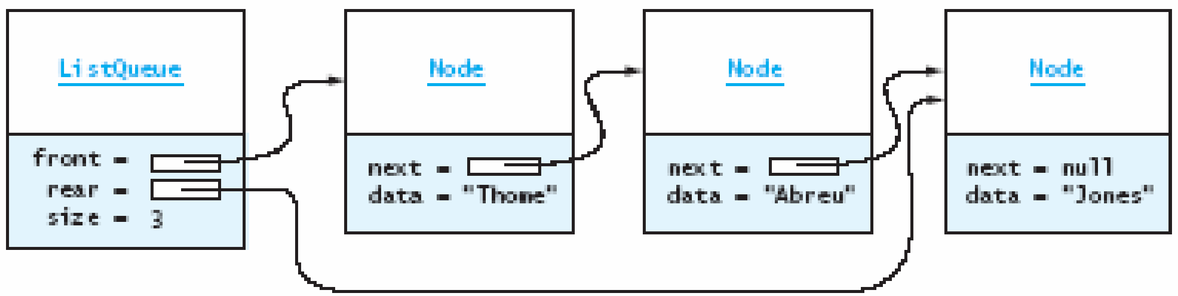
Implementing **SLLQueue** (3)

```
public E peek () {
  return (size == 0) ? null : front.data;
}

// we omit the remaining methods and
// the iterator (we’ve done it before)
```

Using a Single-Linked List to Implement a Queue (continued)

FIGURE 6.6
A Queue as a Single-Linked List



Implementing Queue With Java's LinkedList

- Can implement as adapter of any class that implements the **`List`** interface
  - **`ArrayList`**
  - **`Vector`**
  - **`LinkedList`**
- Removal is O(1) with a **`LinkedList`**
  - O(n) when using **`ArrayList`** or **`Vector`**
- How can we get *space* efficiency of **`ArrayList`**, with the *time* efficiency of **`LinkedList`**?

Analysis of the Space/Time Issues

- Time efficiency of singly- or doubly-linked list good: O(1) for all **Queue** operations
- Space cost: ~3 extra words per item
  - ArrayList uses 1 word per item when fully packed
  - 2 words per item when just grown
  - On average ~1.5 words per item, for larger lists

Analysis of the Space/Time Issues (2)

- **`ArrayList`** Implementation
  - Insertion at end of array is O(1), on average
  - *<u>Removal from the front is linear time: O(n)</u>*
  - Removal from rear of array is O(1)
  - *<u>Insertion at the front is linear time: O(n)</u>*

Implementing **Queue** With a *Circular Array*

Basic idea: Maintain two integer indices into an array

- *front*: index of first element in the queue
- *rear*: index of the last element in the queue
- Elements thus fall at front through read

Key innovation:

- If you hit the end of the array *wrap around* to slot 0
- This prevents our needing to shift elements around

- Still have to deal with overflow of space

Implementing Queue With Circular Array (2)

FIGURE 6.7
A Queue Filled with Characters

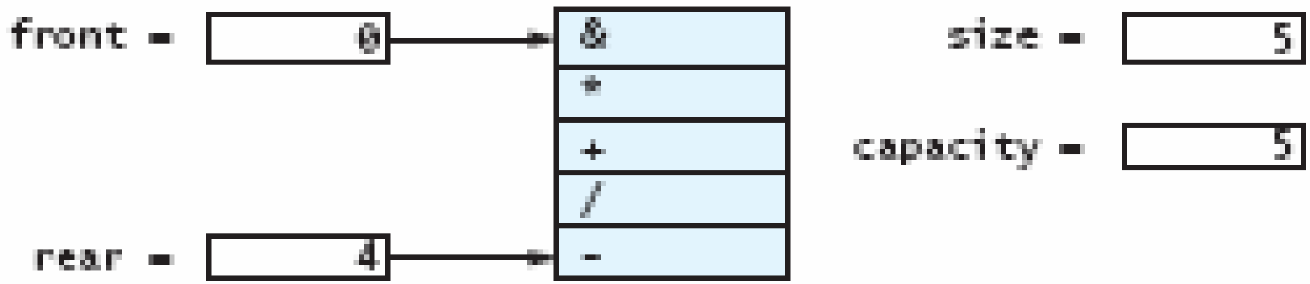
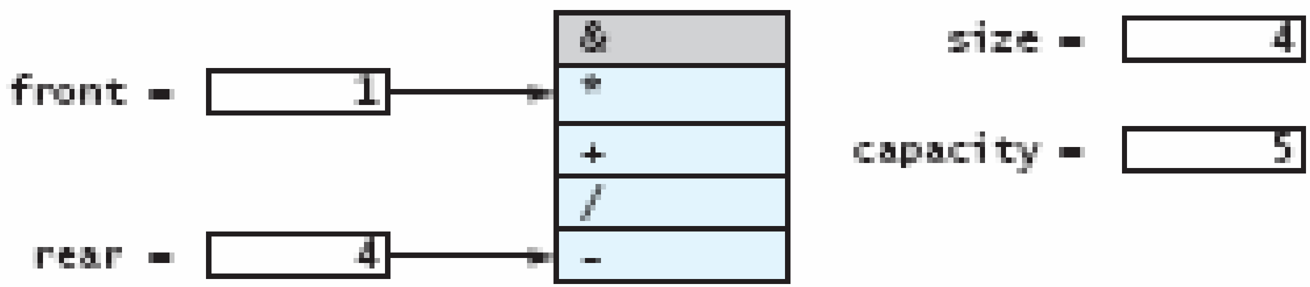
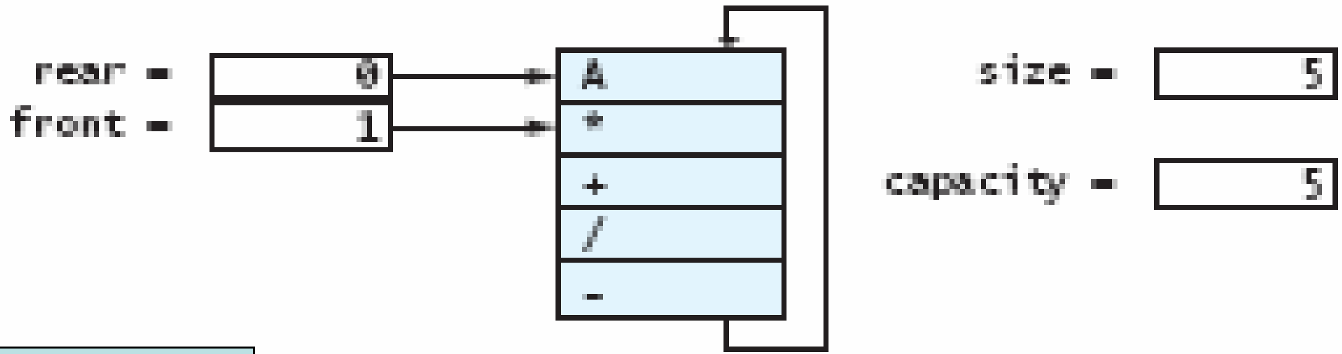


FIGURE 6.8
The Queue after Deletion of the First Element



Implementing Queue With Circular Array (3)

FIGURE 6.9
A Queue as a Circular Array

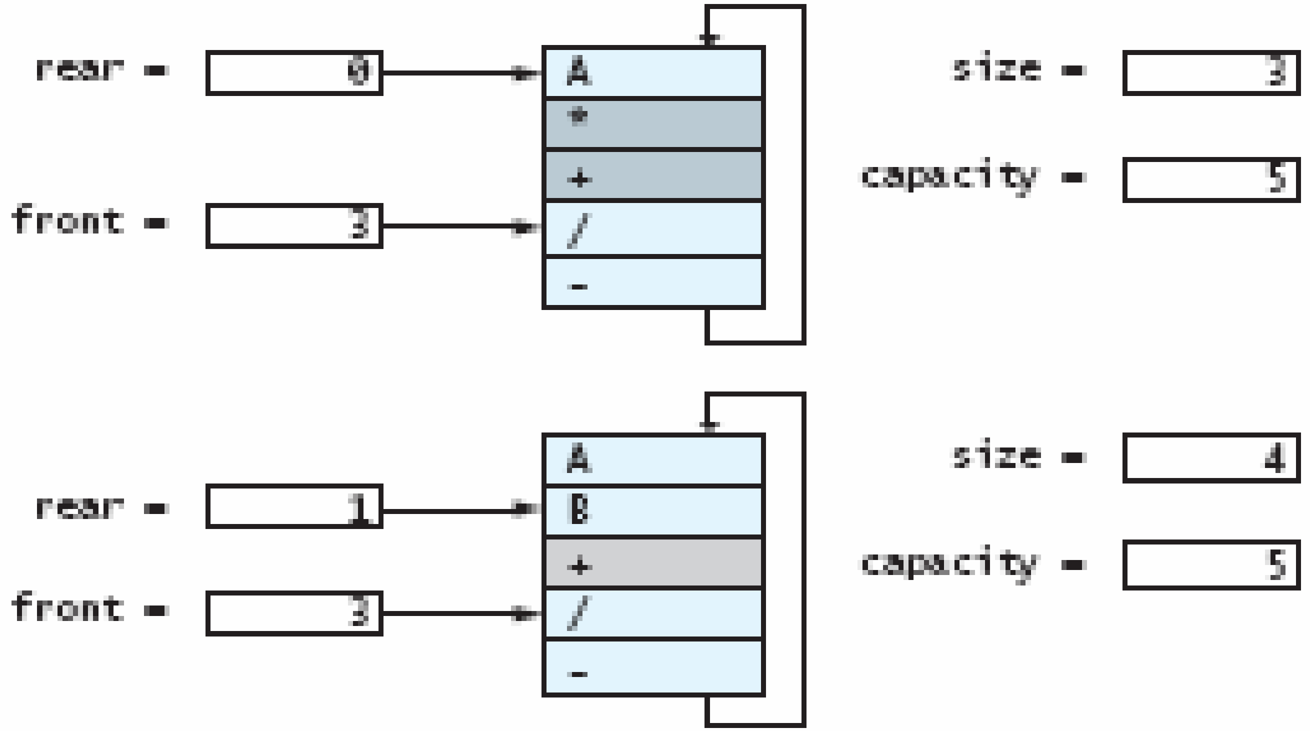


After adding one more element: A

Implementing Queue With Circular Array (4)

FIGURE 6.10
The Effect of Two Deletions . . .

and One Insertion



Implementing **ArrayQueue**

```
public class ArrayQueue<E>
    extends AbstractQueue<E>
    implements Queue<E> {
  private int front;  // index first elt
  private int rear;   // index last elt
  private int size;   // current # elts
  private int capacity; // max elts

  private static final int
      DEFAULT_CAPACITY = ...;
  private E[] data;
```

Implementing **ArrayQueue** (2)

```
public ArrayQueue () {
  this(DEFAULT_CAPACITY);
}

public ArrayQueue (int capacity) {
  data = (E[]) new Object[capacity];
  this.capacity = capacity;
  this.size  = 0;
  this.front = 0;
  this.rear  = capacity - 1;
}
```

Implementing **ArrayQueue** (3)

```
public boolean offer (E e) {
  if (size >= capacity) reallocate();
  size++;
  // statement belows increases by 1,
  // but circularly in array with
  // capacity slots
  rear = (rear + 1) % capacity;
  data[rear] = e;
  return true;
}

// Cost: O(1) (even on average when grows)
```

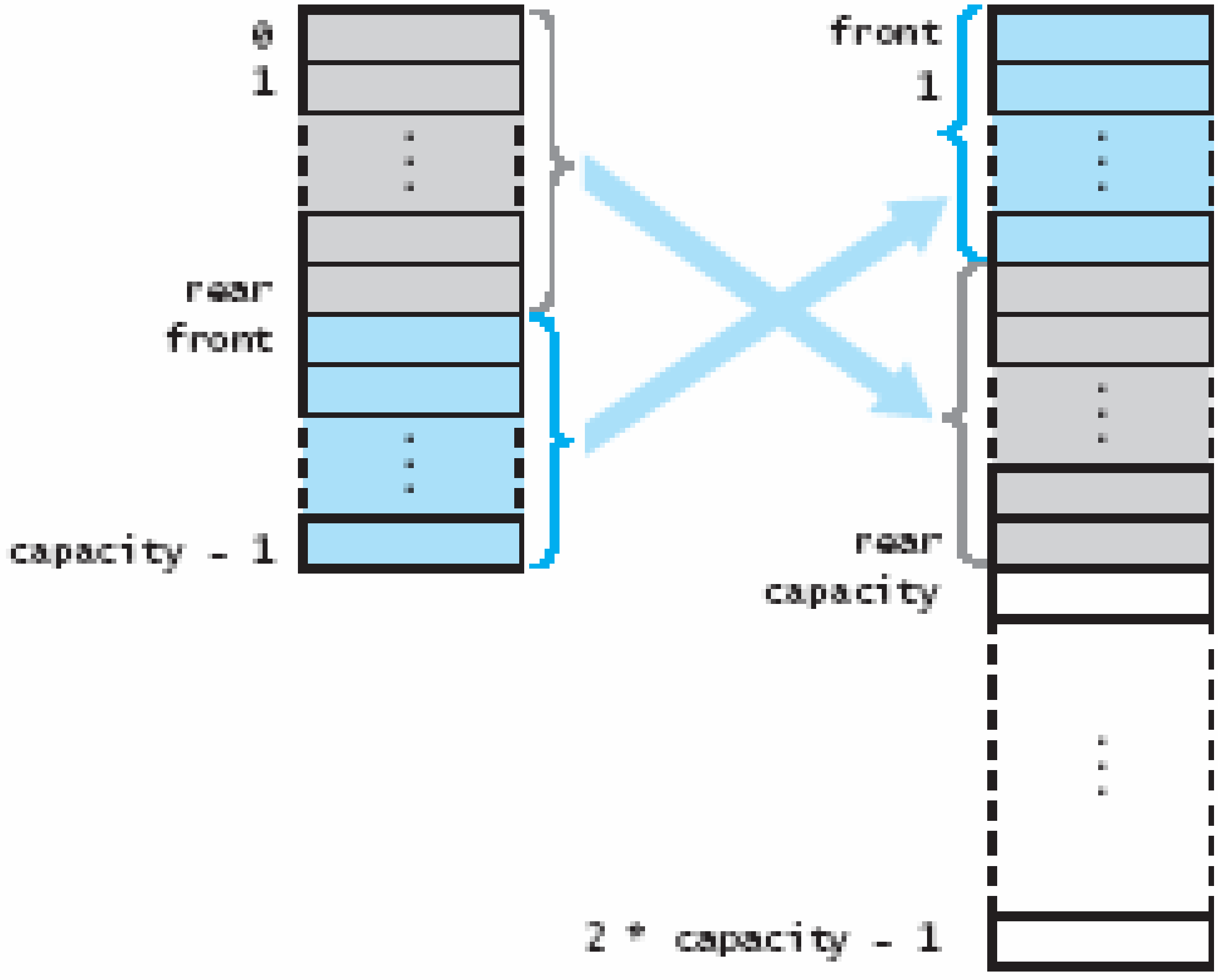
Implementing **ArrayQueue** (4)

```
public E peek () {
  return (size == 0) ? null : data[front];
}

public E poll () {
  if (size == 0) return null;
  E result = data[front];
  front = (front + 1) % capacity;
  size--;
  return result;
}
```

Implementing Queue With Circular Array

FIGURE 6.11
Reallocating a Circular Array



Implementing **ArrayQueue.reallocate**

```
private void reallocate () {
  int newCap = capacity * 2;
  E[] newData = (E[]) new Object[newCap];
  int j = front;
  for (int i = 0; i < size; i++) {
    newData[i] = data[j]
    j = (j + 1) % capacity;
  }
  front = 0;
  rear  = size – 1;
  data  = newData;
  capacity = newCap;
}
```

ArrayQueue.reallocate: Tricky Version

```
// replace for loop with:
if (rear >= front) { // no wrap
  System.arraycopy(data, front,
    newData, 0, size);
} else {  // wraps: copy in two parts
  int firstPart = capacity – front;
  System.arraycopy(data, front,
    newData, 0, firstPart);
  System.arraycopy(data, 0,
    newData, firstPart, front);
}
```

ArrayQueue<E>.Iter (2)

```
public boolean hasNext () {
  return count < size;
}

public E next () {
  if (!hasNext())
    throw new NoSuchElementException();
  E value = data[index];
  index = (index + 1) % capacity;
  count++;
  return value;
}
```

ArrayQueue<E>.Iter (3)

```
public void remove () {
  throw new
    UnsupportedOperationException();
}
```

Comparing the Three Implementations

- All three are comparable in *time*: O(1) operations
- Linked-lists require more storage
  - Singly-linked list: ~3 extra words / element
  - Doubly-linked list: ~4 extra words / element
- Circular array: 0-1 extra word / element
  - On average, ~0.5 extra word / element

Simulating Waiting Lines Using Queues

- Simulation is used to study the performance:
  - Of a physical (“real”) system
  - By using a physical, mathematical, or computer model of the system
- Simulation allows designers to estimate performance
  - *Before* building a system
- Simulation can lead to design improvements
  - Giving *better* expected performance of the system

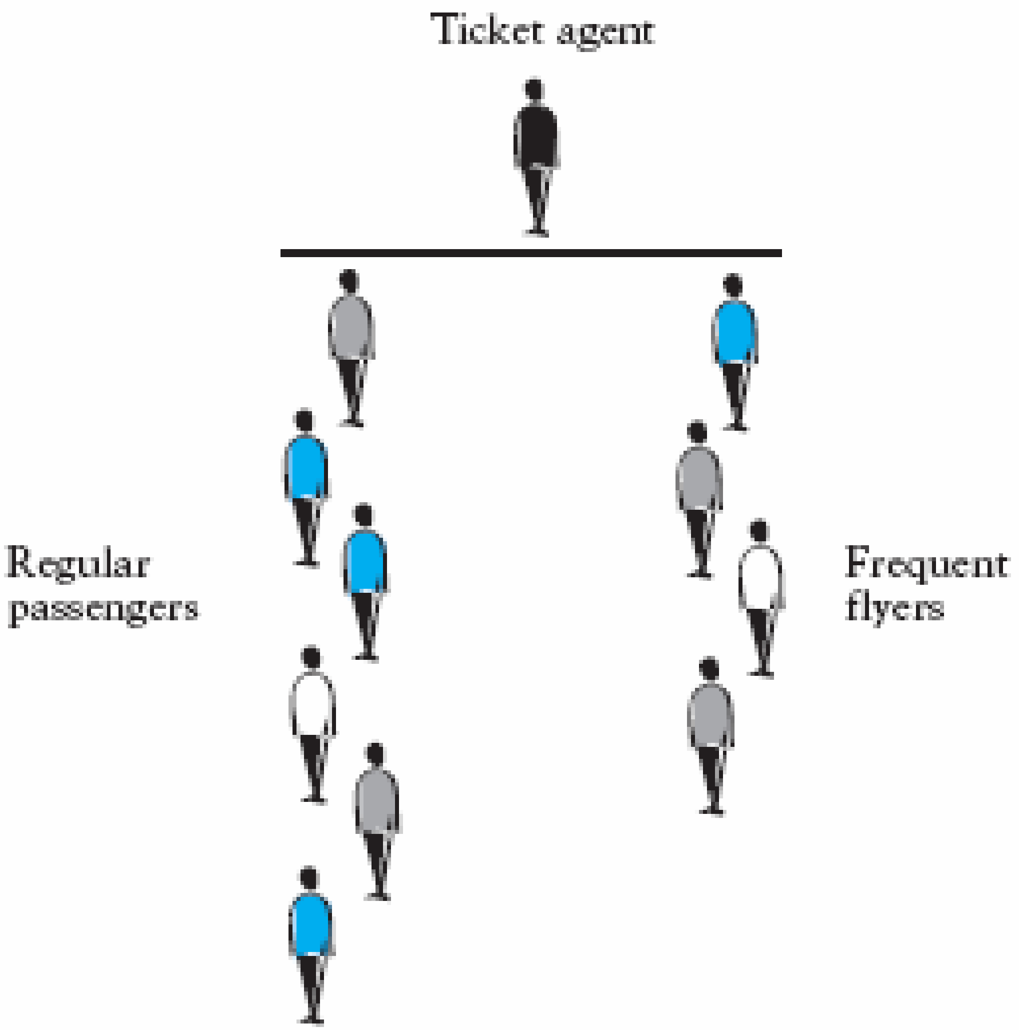
Simulating Waiting Lines Using Queues (2)

- Simulation is particular useful when:
  - Building/changing the system is expensive
  - Changing the system later may be dangerous
- Often use computer models to simulate “real” systems
  - Airline check-in counter, for example
  - Special branch of mathematics for these problems:

    Queuing Theory

Simulate Strategies for Airline Check-In

FIGURE 6.12
Passenger Waiting Lines



Simulate Airline Check-In

- We will maintain a simulated *clock*
  - Counts in integer “ticks”, from 0
- At each tick, one or more events can happen:

1. Frequent flyer (FF) passenger arrives in line
2. Regular (R) passenger arrives in line
3. Agent finishes, then serves next FF passenger
4. Agent finishes, then serves next R passenger
5. Agent is idle (both lines empty)

Simulate Airline Check-In (2)

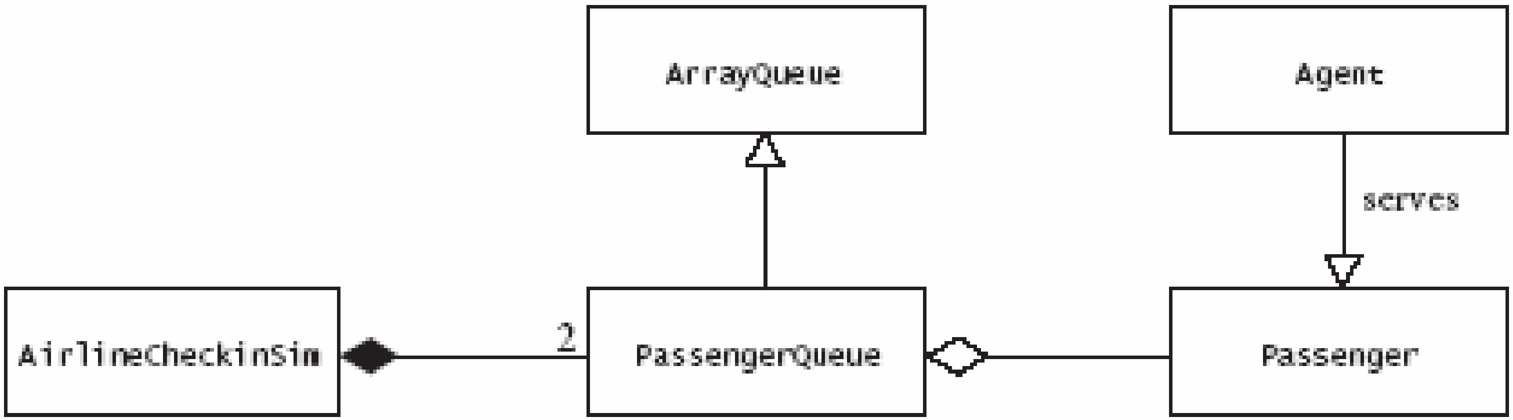
- Simulation uses some *parameters:*
  - Max # FF served between regular passengers
  - Arrival rate of FF passengers
  - Arrival rate of R passengers
  - Service time
- Desired output:
  - Statistics on waiting times, agent idle time, etc.
  - Optionally, a detailed trace

Simulate Airline Check-In (3)

- Design approach:
  - **`Agent`** data type models airline agent
  - **`Passenger`** data type models passengers
  - 2 **`Queue<Passenger>`**, 1 for FF, 1 for R
  - Overall **`CheckinSim`** class

Simulate Airline Check-In (4)

FIGURE 6.13
Airline Check-In Simulation: Initial UML Class Diagram



`CheckinSim` Design

Instance fields:

```
private PsgrQueue freqFlyQueue;
private PsgrQueue regQueue;
private int freqFlyMax; // max between reg
private int maxServeTime; // max agt time
private int totalTime; // length of sim.
private boolean showAll; // show trace?
private int clock; // current time
```

CheckinSim Design (2)

More instance fields:

```
private int timeDone; // curr psgr finish
private int freqFlySinceReg;
```

`CheckinSim` Design (3)

Methods:

```
public static void main (String[] args)
  // enter data, then run simulation
private void runSimulation ()
private void enterData ()
private void startServe ()
  // start to serve a passenger
private void showStats ()
```

PsgrQueue Design

Instance fields:

```
private Queue<Passenger> theQueue;
private int numServed;
private int totalWait;
  // total waiting time for all psgrs
private String queueName;
private double arrivalRate;
```

`PsgrQueue` Design (2)

Methods:

```
public PsgrQueue (String queueName)
private void checkNewArrival (
  int clock, boolean showAll)
private int update ( // update wait time
  int clock, boolean showAll)
public int getTotalWait ()
public int getNumServed ()
```

Passenger Design

Methods:

```
public Passenger (int arrivalTime)
  // arrives at given time, service time
  // uniformly random in 1..maxServeTime
public int getArrivalTime ()
public int getServeTime ()
public static void setMaxServeTime (
  int maxServeTime)
```

CheckinSim Implementation

```
public class CheckinSim {
  private PsgrQueue ffQueue =
    new PsgrQueue(“Frequent Flyer”);
  private PsgrQueue rQueue =
    new PsgrQueue(“Regular Passenger”);
  private int ffMax;
  private int maxServeTime;
  private int totalTime;
  private boolean show;
  private int clk;
  private int timeDone;
  private int ffSinceReg;
  ...
```

`CheckinSim` Implementation (2)

```
public static void main (String[] args) {
  CheckinSim sim = new CheckinSim();
  sim.enterData();
  sim.runSimulation();
  sim.showStats();
  System.exit(0);
}
```

CheckinSim Implementation (3)

```
private void enterData () {
  // interact with user to choose and set
  // values for:
  // - FF queue arrivalRate
  // - Reg queue arrivalRate
  // - maxServeTime
  // - totalTime (length of simulation)
  // - show
  ...
}
```

CheckinSim Implementation (4)

```
private void runSimulation () {
  for (clk = 0; clk < totalTime; ++clk) {
    ffQueue.checkNewArrival(clk, show);
    rQueue .checkNewArrival(clk, show);
    if (clk >= timeDone) startServe();
  }
}
```

CheckinSim Implementation (5)

```
private void startServe () {
  if (!ffQueue.isEmpty() &&
      ((ffSinceReg <= ffMax) ||
       rQueue.isEmpty())) {
    ffSinceReg++;
    timeDone = ffQueue.update(clk, show);
  } else {
    ffSinceReg = 0;
    timeDone = rQueue.update(clk, show);
  } else if (show)
    System.out.println("Time is " + clk +
      " server is idle");
}
```

CheckinSim Implementation (6)

```
private void showStats () {
  // print:
  // # reg psgrs served, average wait
  // # ff psgrs served, average wait
  // # reg psgrs remaining
  // # ff psgrs remaining
}
```

PsgrQueue Implementation

```
public class PsgrQueue {
  private Queue<Passenger> theQueue;
  private int numServed = 0;
  private int totalWait = 0;
  private String queueName;
  private double arrivalRate;

  public PsgrQueue (String queueName) {
    this.queueName = queueName;
    this.theQueue =
      new LinkedList<Passenger> ();
  }
  ...
```

PsgrQueue Implementation (2)

```
public void checkNewArrival (
    int clk, boolean show) {
  if (Math.random() < arrivalRate) {
    // random in [0..1]
    // rate = psgrs/min (< 1)
    theQueue.add(new Passenger(clk));
    if (show)
      System.out.println(“Time is “ +
        clk + “: “ + queueName +
        “ arrival, new queue size is “ +
        theQueue.size())
  }
}
```

PsgrQueue Implementation (3)

```
public int update (
    int clk, boolean show) {
  Passenger psgr = theQueue.remove();
  int time = psgr.getArrivalTime();
  int wait = clk – time;
  totalWait += wait;
  numServed++;
  if (show)
    System.out.println(“Time is “ + clk +
      “: Serving “ + queueName +
      “ with time stamp “ + time);
  return clk + psgr.getServeTime();
}
```

Passenger Implementation

```
public class Passenger {
  private int psgrId;
  private int serveTime;
  private int arrivalTime;
  private static int maxServeTime;
  private static int idNum = 0;
  public Passenger (int arrivalTime) {
    this.arrivalTime = arrivalTime;
    this.serveTime =
      1 + Random.nextInt(maxServeTime);
          // random in 0..maxServeTime-1
    this.psgrId = idNum++;
  }  ...
```

Passenger Implementation (2)

```
public int getArrivalTime () {
  return arrivalTime;
}
public int getServeTime () {
  return serveTime;
}
public int getId () { return psgrId; }

public static void setMaxServeTime (
    int maxServeTime) {
  Passenger.maxServeTime = maxServeTime;
}
```

Concerning “Random” Numbers

- Not *really* random, but *pseudo*-random
  - Generated by a definite algorithm
  - Next produced from the last
  - Thus, sequence determined by starting value
  - Starting value is called the *seed*
  - Seed usually set from date/time, but can set directly to get *same* sequence on each occasion
    - Good for testing!

Concerning "Random" Numbers (2)

- Inside **`java.util.Random`** a 48-bit number for each separate (pseudo)random number sequence/stream
- **`java.lang.Math.random()`** gets two integers, one with 27 "random" bits and one with 26, and combines them to get a 53-bit "random" **`double`** with value in the range [0.0d, 1.0d) (i.e., 0 <= random() < 1)
- **`java.util.Random`** makes it easy to get random numbers over a range [0, n) (0 through n-1), etc.
- All these generators are *uniform*: any value in the range is equally likely at each call (in theory anyway)

A Different Random Number Distribution

- Uniform inter-arrival times are often not the best for modeling queueing systems
- Many distributions are used, but an important one is the *Poisson* distribution
- For values of $t \geq 0$, choose t with probability $\lambda e^{-\lambda t}$
- This models steady arrivals with average rate λ
- Given a random number $r > 0$, compute inter-arrival time as $(-\ln r)/\lambda$
- In Java: **`-log(1.0-Math.random())/lambda`**
- The **`1.0-random()`** insures **`log`**'s argument > 0.0